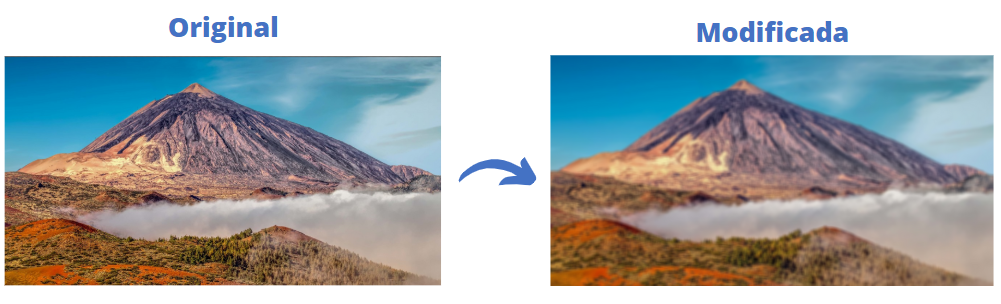


# **Práctica 3**

**1. Elegir cualquier problema de tratamiento de imágenes e implementarlo en C/C++. Puede ser de tratamiento se vídeo.**

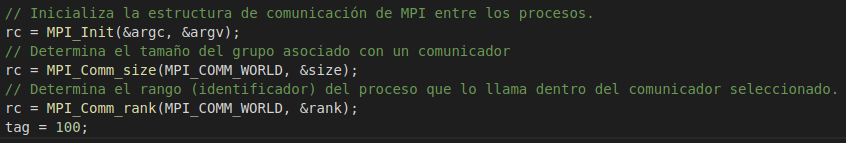
Para este apartado he tratado de buscar un algoritmo secuencial que permita el tratamiento de una imagen. Tras buscar entre varios, me he decantado por el que podemos ver a continuación “[https://gist.github.com/OmarAflak/ aca9d0dc8d583ff5a5dc16ca5cdda86a](https://gist.github.com/OmarAflak/%20aca9d0dc8d583ff5a5dc16ca5cdda86a)”.

El algoritmo comentado con anterioridad nos permite aplicar un filtro gaussiano que nos generará un desenfoque a una imagen. Esto lo podremos ver en la siguiente diapositiva:

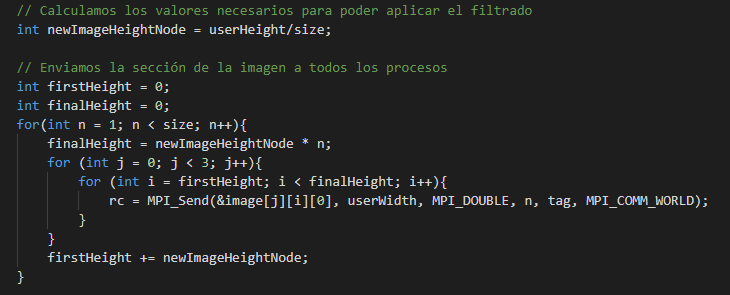
**2. Implementa una versión MPI para este algoritmo.**

Tras entender el código secuencial me he dispuesto a modificarlo para crear una versión usando MPI. Los pasos para lograr esto han sido:

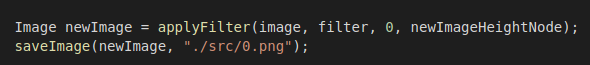
1. Inicializar la estructura de MPI.



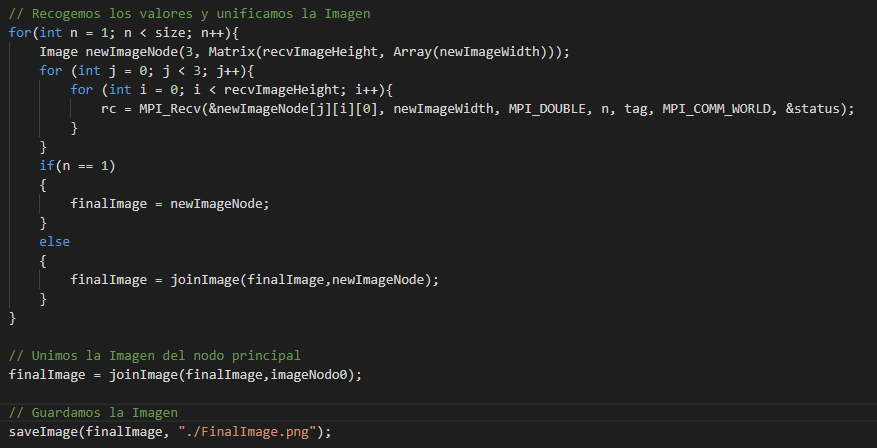
1. Dividir el programa en dos. Uno para si el proceso es el inicial y otro para el resto de los procesos.
2. Dividimos la imagen por el número de proceso. Esto nos ayudará a saber que sección deberá realizar cada proceso. Posteriormente, se enviará la imagen para que cada proceso pueda aplicar el filtrado.



1. Cada proceso (incluido el principal) tendrá que realizar el filtrado de la sección que tiene asignada.



1. Por último, cada proceso enviará cada sección de imagen calculada y el principal las unificará para formar la Imagen final.



Cabe resaltar que se ha modificado la función de aplicar el filtrado debido a que, era necesario que solo realizara una sección de la imagen y devuelva únicamente dicha sección. De igual forma, se ha intentado mejorar el código reduciendo un total de 10s.

Tras realizar todas las modificaciones, se ha compilado el código obteniendo los siguientes resultados que veremos a continuación. En caso de utilizar diferentes tamaños de kernel el programa tarda menos pero el filtro es menor.

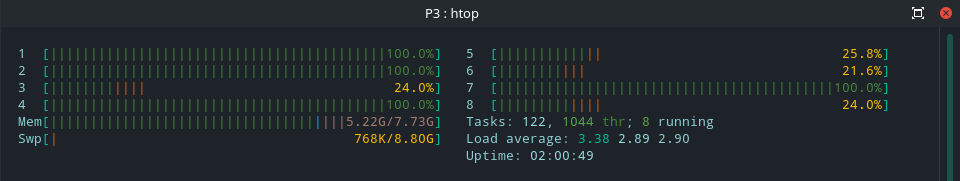
**Programa 10x10 kernel**

****

**Programa kernel 5 x 5**

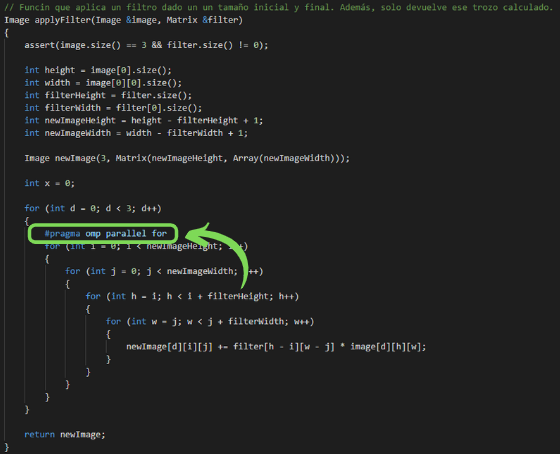
****

**HTOP**



**3. Desarrolla una versión OpenMP para este algoritmo.**

Para poder crear una versión de OpenMP se ha cogido la versión secuencial y se ha modificado para que use todos los hilos posibles dentro de la máquina. Para lograr esto, se ha añadido lo siguiente a la función de aplicar el filtrado:



Tras añadir lo anterior, se ha probado el código obteniendo las diapositivas que veremos a continuación. En caso de utilizar diferentes tamaños de kernel, el programa tarda menos pero el filtro es menor

**Programa 10x10 kernel**

****

**Programa 5x5 kernel**

****

**HTOP**

****

**4. Compara ambas versiones.**

Entre ambas versiones existen diversas diferencias tanto en el tiempo de cómputo, como en la facilidad de la programación. Esto lo podremos ver a continuación:

**Tiempo de Computo**

En cuanto al tiempo de cómputo, el código hecho con MPI es más eficiente que el realizado con OPENMP. Aunque en primera instancia, nos parezca lo contrario hay que tomar en cuenta que la opción de MPI debe descargar la imagen sobre cada proceso y, posteriormente, escribir y leer de un recurso compartido. Esta problemática se podría resolver intentando buscar un método más eficiente.

**Facilidad de programación**

En cuanto a la facilidad de programación la opción OpenMP es mucho más sencillo y rápido de realizar. Esto es debido a que la gran parte de la complejidad de MPI ya se realiza de forma automática (envíos, dividir el problema,…).

**Conclusión**

Por todo lo comentado con anterioridad, opino que ambas tecnologías tienen un uso para una circunstancia. Si se quiere primar la velocidad de programación se deberá de usar OpenMP. Sin embargo, si se quiere priorizar el tiempo de computo se deberá de usar MPI. Esto es debido principalmente a que MPI es más configurable. En cambio, OpenMp está más limitado.